

**DEPARTMENT OF INFORMATION AND COMMUNICATION ENGINEERING**

FACULTY OF ENGINEERING AND TECHNOLOGY  
PRACTICAL LAB REPORT

**Practical**

**Lab**

**Report**Course Code: MATH-2101  
**Course Title: Discrete Mathematics and Numerical Methods Sessional**

**Submitted By: Submitted To:**Name: Monjur Morshed Prof .Md. Anwar Hossain  
Roll: 220601 Professor  
Session: 2021-22 Department of ICE  
Department: ICE, PUST Pabna University of Science and Technology

**Date of Submission:21/04/2025**

|  |  |
| --- | --- |
| **Index** | |
| **SN** | **Problem Description** |
|  | Let A be the set Write a program to find the ordered pairs are in the relation  I) II) |
|  | Suppose that and Let R be the relation from A to B containing (a, b) if , , and . Write a program to find the relation R and also represent this relation in matrix form if , and , and and . |
|  | Write a program for graph coloring by Welch- Powell’s algorithm. |
|  | Write a program to find shortest path by Warshall’s algorithm. |
|  | Suppose that the relations R1 and R2 on a set A are represented by the matrices  and . Write a program to find the MR1∪R2 and MR1∩R2 . |
|  | The following table gives the population of a town during the last six censuses. Write a program to find the population in the year of 1946 using Newton-Gregory forward interpolation formula.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Year: | 1911 | 1921 | 1931 | 1941 | 1951 | 1961 | | Population: | 12 | 15 | 20 | 27 | 39 | 52 | |
|  | Write a program to find ***f(7.5)*** form the following table using Newton-Gregory backward interpolation formula.   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | *x*: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | *f(x)*: | 1 | 8 | 27 | 64 | 125 | 216 | 343 | 512 | |
|  | Write a program to find the value of f(15) from the following table using Newton’s divided difference formula for unequal intervals.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | *x:* | 4 | 5 | 7 | 10 | 11 | 13 | | *f(x):* | 48 | 100 | 294 | 900 | 1210 | 2028 | |
|  | The values of y and x are given as below:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | *x:* | 5 | 6 | 9 | 11 | | *y:* | 12 | 13 | 14 | 16 |   Write a program to find the value of y when x=10 using Lagrange’s interpolation formula for unequal intervals. |
|  | Write a program to find a real root of the equation that lies between -2 and -1.5 using bisection method. |
|  | Write a program to find a root of the function in the range 1<x<3 using false position method. |

**Experiment no.1**

**Experiment Name: Let A be the set Write a program to find the ordered pairs are in the relation**

**I) II)**

**Objective:**

To find all ordered pairs in the relations R1 = {(a, b) | a divides b} and R2 = {(a, b) | a ≤ b} for the set A = {1, 2, 3, 4}.

**Theory:**

Relation R1 (a∣b)*R*1 (*a*∣*b*):

* A relation R on set *A* is a subset of A×A.
* Definition: (a,b)∈R1 if a*a* divides b*b*, i.e., k∈Z such that b=a⋅k.
* Example: For A={1,2,3,4}*,* R1={(1,1),(1,2),(1,3),(1,4),(2,2),(2,4),(3,3),(4,4)}*.*

Relation R2 (a≤b)*R*2 (*a*≤*b*):

* Definition: (a,b)∈R2if a≤b.
* Properties: Reflexive, antisymmetric, and transitive (a partial order).
* Example: R2={(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)}*.*

**Algorithm:**

1. Define the set A = {1, 2, 3, 4}
2. For each pair (a, b) where a, b ∈ A:
   * For R1: Check if a divides b (b % a = 0)
   * For R2: Check if a ≤ b
3. Output all pairs that satisfy the respective relation

**Code(C++):**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> A = {1,2,3,4};

cout << "R1: ";

for (int a : A)

for (int b : A)

if (b % a == 0) cout << "(" << a << "," << b << ") ";

cout << "\nR2: ";

for (int a : A)

for (int b : A)

if (a <= b) cout << "(" << a << "," << b << ") ";

return 0;

}

**Output / Result:**

R1: (1,1) (1,2) (1,3) (1,4) (2,2) (2,4) (3,3) (4,4)

R2: (1,1) (1,2) (1,3) (1,4) (2,2) (2,3) (2,4) (3,3) (3,4) (4,4)

**Experiment no.2**

**Experiment Name: Suppose that and Let R be the relation from A to B containing (a, b) if , , and . Write a program to find the relation R and also represent this relation in matrix form if , and , and and .**

**Objective:**

To find the relation R from set A = {1, 2, 3} to set B = {1, 2} containing pairs (a, b) where a > b, and represent this relation in matrix form.

**Theory:**

* Relation *R*: R⊆A×B, where A={1,2,3}*,*B={1,2}*,* and (a,b)∈R if a>b
* Matrix Representation:
  + Rows correspond to elements of A, columns to elements of B.
  + Entry M[i][j]=1if ai>bj​, else 0.
  + Example Matrix:

**Algorithm:**

* Define sets A = {1, 2, 3} and B = {1, 2}
* Initialize a |A| × |B| matrix (3×2) with all entries as 0
* For each a ∈ A and b ∈ B:
* If a > b, add (a, b) to relation R and set the corresponding matrix entry to 1
* Output the relation R and its matrix representation.

**Code (C++):**

#include <iostream>

using namespace std;

int main() {

int A[] = {1,2,3}, B[] = {1,2};

int matrix[3][2] = {0};

cout << "Pairs: ";

for (int a : A)

for (int b : B)

if (a > b) {

cout << "(" << a << "," << b << ") ";

matrix[a-1][b-1] = 1;

}

cout << "\nMatrix:\n";

for (int i=0; i<3; i++) {

for (int j=0; j<2; j++)

cout << matrix[i][j] << " ";

cout << endl;

}

return 0;

}

**Output / Result:**

Pairs: (2,1) (3,1) (3,2)

Matrix:

0 0

1 0

1 1

**Experiment no.3**

**Experiment Name: Write a program for graph coloring by Welch- Powell’s algorithm.**

**Objective:**

To color the vertices of an undirected graph using the **Welsh-Powell algorithm** such that:

* No two adjacent vertices have the same color.
* The total number of colors used is minimized.

**Theory:**

**Graph Coloring** is an assignment of colors to vertices of a graph such that no two adjacent vertices share the same color.

**Welsh-Powell Algorithm:**

* A greedy graph coloring algorithm.
* Vertices are colored in order of decreasing degree.
* Each vertex is assigned the lowest possible color not used by its neighbors.

**Algorithm:**

1. Represent the graph using an adjacency matrix.
2. Count the degree of each vertex.
3. Sort vertices in descending order of degree.
4. Initialize all vertex colors as uncolored (-1).
5. For each uncolored vertex:
   * Assign it the current color.
   * Assign the same color to any other uncolored vertex that is not adjacent to any already-colored vertex.
6. Repeat until all vertices are colored.

**Code(C++):**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

    vector<vector<int>> adj = {{0,1,1,0}, {1,0,1,1}, {1,1,0,1}, {0,1,1,0}};

    vector<pair<int, int>> degrees;

    for (int i=0; i<4; i++) {

        int deg = 0;

        for (int j=0; j<4; j++) deg += adj[i][j];

        degrees.push\_back({i, deg});

    }

    sort(degrees.begin(), degrees.end(), [](auto &a, auto &b) { return a.second > b.second; });

    vector<int> color(4, -1);

    for (auto &p : degrees) {

        int v = p.first;

        if (color[v] != -1) continue;

        vector<bool> available(4, true);

        for (int u=0; u<4; u++)

            if (adj[v][u] && color[u] != -1) available[color[u]] = false;

        int cr = 0;

        while (!available[cr]) cr++;

        color[v] = cr;

    }

    cout << "Colors: ";

    for (int c : color) cout << c+1 << " ";

    return 0;

}

**Output / Result:**

Colors: 3 1 2 3

**Experiment no.4**

**Experiment Name: Write a program to find shortest path by Warshall’s algorithm.**

**Objective:**

To find the shortest path between all pairs of vertices in a weighted graph using the Floyd-Warshall Algorithm.

**Theory:**

The Floyd-Warshall algorithm is a dynamic programming technique used to find shortest paths between all pairs of vertices in a weighted graph (both directed and undirected).

* Works with positive and negative weights.
* Cannot handle negative cycles.

Core idea:  
Iteratively update the shortest path between every pair of vertices (i, j) by checking if there's a shorter path through an intermediate vertex k.

Formula:

**Algorithm:**

1. Initialize the distance matrix with graph weights.
2. For each intermediate vertex k from 0 to V-1:
   * For each pair (i, j), check if path i → k → j is shorter than i → j.
3. Update dist[i][j] accordingly.
4. Print the final shortest distance matrix.

**Code (C++):**

#include <iostream>

using namespace std;

#define INF 99999

int main()

{

int dist[4][4] = {{0, 5, INF, 10}, {INF, 0, 3, INF}, {INF, INF, 0, 1}, {INF, INF, INF, 0}};

for (int k = 0; k < 4; k++)

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

cout << "Shortest Paths:\n";

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

(dist[i][j] == INF) ? cout << "INF " : cout << dist[i][j] << " ";

cout << endl;

}

return 0;

}

**Output / Result:**

Shortest Paths:

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Experiment no.5**

**Experiment Name: Suppose that the relations R1 and R2 on a set A are represented by the matrices**

**and . Write a program to find the MR1∪R2 and MR1∩R2 .**

**Objective:**

To perform **union** and **intersection** operations on two **relation matrices** and print the resulting matrices.

**Theory:**

Relations can be represented using matrices where entry (i, j) is 1 if the pair (i, j) is in the relation, and 0 otherwise.

For two relations R1 and R2 represented by matrices MR1 and MR2:

* The union R1 ∪ R2 contains all pairs that are in either R1 or R2 or both
* The intersection R1 ∩ R2 contains all pairs that are in both R1 and R2

Matrix operations for relations:

* Union (MR1∪R2): MR1∪R2[i][j] = 1 if MR1[i][j] = 1 OR MR2[i][j] = 1, otherwise 0
* Intersection (MR1∩R2): MR1∩R2[i][j] = 1 if MR1[i][j] = 1 AND MR2[i][j] = 1, otherwise 0

**Equation:**

**Union** (R1∪R2*R*1∪*R*2):

**Intersection** (R1∩R2*R*1∩*R*2):

**Algorithm:**

 Define the relation matrices MR1 and MR2 as given in the problem

 For union (R1 ∪ R2):

* Create a new matrix with the same dimensions
* For each position (i, j), set the value to 1 if either MR1[i][j] = 1 or MR2[i][j] = 1 (logical OR)

 For intersection (R1 ∩ R2):

* Create a new matrix with the same dimensions
* For each position (i, j), set the value to 1 if both MR1[i][j] = 1 and MR2[i][j] = 1 (logical AND)

 Display the resulting matrices

**Code (C++):**

#include <iostream>

using namespace std;

int main()

{

    int MR1[3][3] = {{1, 0, 1}, {1, 0, 0}, {0, 1, 0}};

    int MR2[3][3] = {{1, 0, 1}, {0, 1, 1}, {1, 0, 0}};

    cout << "Union:\n";

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 3; j++)

            cout << (MR1[i][j] || MR2[i][j]) << " ";

        cout << endl;

    }

    cout << "Intersection:\n";

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 3; j++)

            cout << (MR1[i][j] && MR2[i][j]) << " ";

        cout << endl;

    }

    return 0;

}

**Output / Result:**

**Union:**

**1 0 1**

**1 1 1**

**1 1 0**

**Intersection:**

**1 0 1**

**0 0 0**

**0 0 0**

**Experiment no.6**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Experiment Name: The following table gives the population of a town during the last six censuses. Write a program to find the population in the year of 1946 using Newton-Gregory forward interpolation formula.**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Year:** | **1911** | **1921** | **1931** | **1941** | **1951** | **1961** | | **Population:** | **12** | **15** | **20** | **27** | **39** | **52** | |

**Objective**

To find the population of a town in the year 1946 using Newton-Gregory forward interpolation formula, given the population data from six censuses.

**Theory**

Newton-Gregory forward interpolation is used to estimate a value of a function at a point between given data points. It is particularly useful when the data points are equally spaced.

The Newton-Gregory forward interpolation formula is:

Where:

* x₀ is the first point in the dataset
* u = (x - x₀)/h, where h is the step size
* Δf, Δ²f, Δ³f, ... are the forward differences

**Algorithm**

1. Calculate the step size h (difference between consecutive x values)
2. Compute the forward difference table for the given data points
3. Calculate u = (x - x₀)/h for the desired interpolation point x
4. Apply the Newton-Gregory forward interpolation formula
5. Return the interpolated value

**Code(C++):**

#include <iostream>

using namespace std;

int main()

{

    int x[] = {1911, 1921, 1931, 1941, 1951, 1961};

    int y[] = {12, 15, 20, 27, 39, 52};

    double u = (1946 - x[3]) / 10.0;

    double diff[6][6];

    for (int i = 0; i < 6; i++)

        diff[i][0] = y[i];

    for (int j = 1; j < 6; j++)

        for (int i = 0; i < 6 - j; i++)

            diff[i][j] = diff[i + 1][j - 1] - diff[i][j - 1];

    double res = y[3] + u \* (diff[3][1] + (u - 1) \* (diff[3][2] / 2 + (u - 2) \* (diff[3][3] / 6 + (u - 3) \* (diff[3][4] / 24))));

    cout << "Population in 1946: " << res << endl;

    return 0;

}

**Output/Result:**

Population in 1946: 32.875

**Experiment no.7**

**Experiment Name: Write a program to find *f(7.5)* form the following table using Newton-Gregory backward interpolation formula.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***x*:** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| ***f(x)*:** | **1** | **8** | **27** | **64** | **125** | **216** | **343** | **512** |

**Objective**

To find the value of f(7.5) using Newton-Gregory backward interpolation formula from a given set of data points.

**Theory**

Newton-Gregory backward interpolation is similar to forward interpolation but works from the end of the data set backward. It's especially useful when we want to find a value near the end of the given data points.

The Newton-Gregory backward interpolation formula is:

Where:

* xn is the last point in the dataset
* u = (x - xn)/h, where h is the step size
* ∇f, ∇²f, ∇³f, ... are the backward differences

**Algorithm**

1. Calculate the step size h (difference between consecutive x values)
2. Compute the backward difference table for the given data points
3. Calculate u = (x - xn)/h for the desired interpolation point x
4. Apply the Newton-Gregory backward interpolation formula
5. Return the interpolated value

**Code(C++):**

#include <iostream>

using namespace std;

int main()

{

    int x[] = {1, 2, 3, 4, 5, 6, 7, 8};

    int y[] = {1, 8, 27, 64, 125, 216, 343, 512};

    double u = (7.5 - 8) / 1.0;

    double diff[8][8];

    for (int i = 0; i < 8; i++)

        diff[i][0] = y[i];

    for (int j = 1; j < 8; j++)

        for (int i = 7; i >= j; i--)

            diff[i][j] = diff[i][j - 1] - diff[i - 1][j - 1];

    double res = y[7] + u \* (diff[7][1] + (u + 1) \* (diff[7][2] / 2 + (u + 2) \* (diff[7][3] / 6)));

    cout << "f(7.5) = " << res << endl;

    return 0;

}

Output:

f(7.5) = 421.875

**Experiment no.8**

**Experiment Name: Write a program to find the value of f(15) from the following table using Newton’s divided difference formula for unequal intervals.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***x:*** | **4** | **5** | **7** | **10** | **11** | **13** |
| ***f(x):*** | **48** | **100** | **294** | **900** | **1210** | **2028** |

**Objective**

To find the value of f(15) using Newton's divided difference formula for unequal intervals, given a set of data points.

**Theory**

Newton's divided difference interpolation is a generalization of the Newton forward and backward interpolation formulas that works for data points with unequal intervals.

The Newton's divided difference formula is:

And

Where f[x₀,x₁,...,xₙ] represents the nth divided difference, defined recursively as:

* First divided difference: f[xᵢ,xᵢ₊₁] = (f(xᵢ₊₁) - f(xᵢ))/(xᵢ₊₁ - xᵢ)
* Higher divided differences: f[xᵢ,xᵢ₊₁,...,xᵢ₊ₙ] = (f[xᵢ₊₁,...,xᵢ₊ₙ] - f[xᵢ,...,xᵢ₊ₙ₋₁])/(xᵢ₊ₙ - xᵢ)

**Algorithm**

1. Compute the divided difference table for the given data points
2. Apply Newton's divided difference formula to calculate the interpolated value
3. Return the interpolated value for the given x

**Code(C++):**

#include <iostream>

using namespace std;

int main()

{

    double x[] = {4, 5, 7, 10, 11, 13};

    double y[] = {48, 100, 294, 900, 1210, 2028};

    double diff[6][6];

    int n = 6;

    for (int i = 0; i < n; i++)

        diff[i][0] = y[i];

    for (int j = 1; j < n; j++)

        for (int i = 0; i < n - j; i++)

            diff[i][j] = (diff[i + 1][j - 1] - diff[i][j - 1]) / (x[i + j] - x[i]);

    double res = diff[0][0];

    double term = 1.0;

    for (int j = 1; j < n; j++)

    {

        term \*= (15 - x[j - 1]);

        res += term \* diff[0][j];

    }

    cout << "f(15) = " << res << endl;

    return 0;

}

**Output/Result:**

f(15) = 3150

**Experiment no.9**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Experiment Name: The values of y and x are given as below:**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | ***x:*** | **5** | **6** | **9** | **11** | | ***y:*** | **12** | **13** | **14** | **16** |   **Write a program to find the value of y when x=10 using Lagrange’s interpolation formula for unequal intervals.** |

**Objective**

To find the value of y when x = 10 using Lagrange's interpolation formula for unequal intervals, given a set of data points.

**Theory**

Lagrange interpolation is a method of polynomial interpolation where we construct a polynomial that passes through all the given data points. Unlike Newton's methods, Lagrange's method doesn't require calculating a difference table.

The Lagrange interpolation formula is:

Each Lᵢ(x) has the property that it equals 1 at x = xᵢ and 0 at all other data points.

**Algorithm**

1. For each data point (xᵢ, yᵢ):
   * Calculate the Lagrange basis polynomial Lᵢ(x)
   * Multiply Lᵢ(x) by yᵢ
2. Sum all these products to get the interpolated value P(x)
3. Return the interpolated value for the given x

**Code(C++):**

#include <iostream>

using namespace std;

int main()

{

    double x[] = {5, 6, 9, 11};

    double y[] = {12, 13, 14, 16};

    double xi = 10, res = 0;

    for (int i = 0; i < 4; i++)

    {

        double term = y[i];

        for (int j = 0; j < 4; j++)

            if (j != i)

                term \*= (xi - x[j]) / (x[i] - x[j]);

        res += term;

    }

    cout << "y(10) = " << res << endl;

    return 0;

}

**Output/Result:**

y(10) = 14.6667

**Experiment no.10**

**Experiment Name: Write a program to find a real root of the equation that lies between -2 and -1.5 using bisection method.**

**Objective**

To find a real root of the equation x² - 4x - 10 = 0 that lies between -2 and -1.5 using the bisection method.

**Theory**

The bisection method is a root-finding algorithm that repeatedly bisects an interval and selects the subinterval in which the root must lie. It's based on the Intermediate Value Theorem: if a continuous function changes sign over an interval, then it has at least one root in that interval.

Steps in the bisection method:

1. Start with an interval [a, b] where f(a) and f(b) have opposite signs
2. Compute the midpoint c = (a + b) / 2
3. If f(c) is close enough to zero, return c as the root
4. Otherwise, determine which half-interval contains the root:
   * If f(a) and f(c) have opposite signs, the root is in [a, c]
   * Otherwise, the root is in [c, b]
5. Repeat steps 2-4 until convergence

Equation:

**Algorithm**

1. Define the function f(x) = x² - 4x - 10
2. Initialize the interval [a, b] = [-2, -1.5]
3. Verify that f(a) and f(b) have opposite signs
4. Compute c = (a + b) / 2
5. If |f(c)| < tolerance or iterations > maxIterations, return c as the root
6. Otherwise, update the interval based on the signs of f(a) and f(c)
7. Repeat steps 4-6 until convergence

**Code(C++):**

#include <iostream>

#include <cmath>

using namespace std;

double f(double x) { return x \* x - 4 \* x - 10; }

int main()

{

    double a = -2, b = -1.5, c;

    while (fabs(b - a) > 0.0001)

    {

        c = (a + b) / 2;

        f(c) \* f(a) < 0 ? b = c : a = c;

    }

    cout << "Root: " << c << endl;

    return 0;

}

Result:

Root: -1.74164

**Experiment no.11**

**Experiment Name: Write a program to find a root of the function in the range 1<x<3 using false position method.**

**Objective**

To find a root of the function x² - x - 2 = 0 in the range 1 < x < 3 using the false position (regula falsi) method.

**Theory**

The false position method, also known as the regula falsi method, is a root-finding algorithm that combines aspects of the bisection method and the secant method. Like the bisection method, it brackets the root in an interval where the function changes sign. However, instead of using the midpoint, it uses linear interpolation to estimate the root.

The formula for calculating the next approximation in the false position method is:

Where [a, b] is the current interval. This formula represents the x-coordinate of the point where the line through (a, f(a)) and (b, f(b)) crosses the x-axis.

**Algorithm**

1. Define the function f(x) = x² - x - 2
2. Initialize the interval [a, b] = [1, 3]
3. Verify that f(a) and f(b) have opposite signs
4. Compute c using the false position formula
5. If |f(c)| < tolerance or iterations > maxIterations, return c as the root
6. Otherwise, update the interval based on the signs of f(a) and f(c)
7. Repeat steps 4-6 until convergence

**Code(C++):**

#include <iostream>

#include <cmath>

using namespace std;

double f(double x) { return x\*x - x - 2; }

int main() {

    double a = 1.0, b = 3.0, c;

    const double epsilon = 1e-6;

    do {

        c = (a \* f(b) - b \* f(a)) / (f(b) - f(a));

        if (f(c) \* f(a) < 0)

            b = c;

        else

            a = c;

    } while (abs(f(c)) > epsilon);

    cout << "Root: " << c << endl;

    return 0;

}

**Output/Result:**

Root: 2